Simon Pepping

xml

Docbook In ConTEXt, a ConTEXt XML mapping for DocBook documents

What is Docbook In ConTEXt?

Docbook In ConTEXt combines two technologies that are widely used by authors of technical literature: the Docbook DTD and the ConTEXt macro package for TEX.

It is a ConTEXt module that allows one to produce a typeset version of a Docbook XML file, in dvi or pdf format.

It takes a Docbook XML file as input for TEX. ConTEXt's built-in XML parser parses the file and applies ConTEXt commands when it reads opening and closing tags. Which ConTEXt commands are applied, and therefore how the output is formatted, is determined by the Docbook In ConTEXt module.

Docbook

Docbook[1] has been available since the early 1990s. Over the years it has evolved into an extensive DTD for technical literature. Long ago extensive, customizable stylesheets[2] became available, first in DSSSL, later also in XSLT. The Jade program and the JadeTEX macro package made it possible to run the DSSSL style sheets and print the output with high-quality free tools. This enabled one to author, format and print Docbook documents without expensive software tools. With the advent of XML and XSLT more free tools have become available.

These combined features have made the Docbook DTD the DTD of choice for technical literature. The Linux Documentation Project is one well-known project that switched over from a private DTD to the Docbook DTD. Due to this strong position, the toolset for working with Docbook documents is growing rapidly, see e.g. `http://www.miwie.org/docbookinfo.html`[3].

How did it start and where is it now?

During EuroTEX 2001 in Kerkrade I had become interested in using ConTEXt because of the beautiful presentation styles used by Hans Hagen and several other speakers. While I was following the ConTEXt email list, I also became interested in ConTEXt's XML capabilities. These seemed so wonderful to me, that I *had* to understand how this could be done using TEX macro programming. I started asking questions. Sometimes Hans answers such questions with the suggestion that one take up some or other project. So he suggested that I start an XML mapping for Docbook.

I really had other plans, but I was so intrigued with ConTEXt's XML capabilities that I could not resist and gave it a start. As an added benefit, I would become more familiar with the Docbook DTD. When I started I certainly was aware that this would not be a small task. Docbook is such a large DTD, allowing its authors to use the hundreds of

1. `http://www.oasis-open.org/docbook/`
2. `http://sourceforge.net/projects/docbook`
3. `http://www.miwie.org/docbookinfo.html`

elements in innumerable combinations. But only while the project evolved did it become evident to me how large it really is.

Michael Wiedmann, who is interested in all possible tools to render Docbook documents, heard about the project soon after I started it. He made several contributions. His support and interest helped me to continue through the difficult phase when a project is no longer new, but you do not yet have anything really usable and you know all too well how much work still has to be done.

Now, a year later, I have some sort of an answer as to how it is possible to program ConTEXt's XML capabilities in TEX macros: Theoretically TEX macro programming is complete (is it called NP-complete?). Hans Hagen is one of the few programmers who can turn this theory into practice.

I also have a working XML mapping for DocBook documents in ConTEXt, which I call Docbook In ConTEXt (DIC). It contains good layout instructions for a number of oft used elements in their more common combinations.

Running Docbook In ConTEXt

Before one can typeset an XML file `myfile.xml`, one should create a TEX driver file `myfile.tex`, which should look something like this:

```
\input xtag-docbook

\starttext
\processXMLfilegrouped{\jobname.xml}
\stoptext
```

Then TEX is invoked as: `texexec myfile.tex` to get a dvi file, or as `texexec -pdf myfile.tex` to get a pdf file.

In the driver file `xtag-docbook` is the file name of the module. The XML document is input with the `\processXMLfilegrouped` command. The filename `\jobname.xml` is always correct provided the driver file and the XML file have the same base name.

Alternatively, one can always use the same driver file, in which the name of the XML file is changed each time.

The ConTEXt documentation indicates that one can also run the XML file as `texexec -xmlfilter=docbook testxml.xml`. This will not work because the name of the Docbook In ConTEXt module does not conform to ConTEXt's naming conventions. It works if the module is renamed as `xtag-doc.tex`.

Customizing Docbook In ConTEXt

A Docbook XML document is a normal ConTEXt document. The commands that make up a ConTEXt document are also at work when a Docbook XML document is processed. They are just one layer away from what the user sees. Therefore the output can be customized as for any ConTEXt document with ConTEXt's setup commands. The setup commands should be given *after* the Docbook In ConTEXt module has been read, so that they override the default setup commands in the module. If you do not give additional setup commands, ConTEXt's defaults are applied. This is an example of a driver file with ConTEXt setup commands:

```
\input xtag-docbook
\setupindenting[medium]
\setupheadertexts[section][pagenumber]
\setupheader[leftwidth=.7\hsize,style=slanted]
\setuppagenumbering[location=]
\setupitemize[each][packed][before=,after=,indentnext=no]
```

```
\starttext
\processXMLfilegrouped{\jobname.xml}
\stoptext
```

Docbook In ConTEXt defines a few setup commands and other customizations of its own.

Section blocks

ConTEXt always applies pagebreaks around section blocks, and it treats the Table of Contents and the Index as chapters. This behaviour can be changed with the `pagebreaks` option of the `\setupXMLDB` command:

- `\setupXMLDB[pagebreaks=all]`: Default ConTEXt behaviour.
- `\setupXMLDB[pagebreaks=sectionblocks]`: ToC and Index do not start a new page, and they are treated as sections. All other section blocks retain their default ConTEXt behaviour.
- `\setupXMLDB[pagebreaks=none]`: In addition to the `sectionblocks` option, bodymatter, appendices and backmatter do not start a new page.

Titles

Titles are formatted with a command of the form `XMLDB\XMLparent title`, where `\XMLDBparent` should be replaced with the name of the element to which the title belongs, e.g. `XMLDBarticletitle`. These commands can be redefined. They take one argument, the title. For example, the article title could be redefined as:

```
\def\XMLDBarticletitle#1%
   {\startalignment[left]\bfb #1\stopalignment \blank}
```

The section titles can be customized with ConTEXt's usual `\setuphead` command.

blockquote, epigraph and attribution

The elements `epigraph` and `blockquote` have their own setup commands `\setupepigraph` and `\setupblockquote`, which have the following options:

- `narrower`. Both `epigraph` and `blockquote` are formatted using ConTEXt's narrower environment. The value of this option is a list of `left`, `right` and `middle` that is passed on to the `\startnarrower` command. See the ConTEXt documentation for `\startnarrower` for the effect of these settings.
- `quote`. The value is `on` or `off`. When on, quotation marks are applied as with ConTEXt's quotation environment.
- `command`. The value is a command or set of commands, which are applied at the start of the `narrower` environment.

The element `attribution` is customized with the command `\setupattribution`, which has one option: `command`. The value is applied at the start of the attribution.

More customizations

Customization has only recently obtained the attention it deserves. More setup commands like those for `blockquote` and `epigraph` will follow. The distribution contains a document `Customization.xml` which will contain an up to date description of the customization options.

Other tools for the same task

Docbook In ConTEXt is not the only tool for typesetting a Docbook document. The canonical tool for typesetting any XML file is XSL + FO. An XSL stylesheet is used to define the desired output in terms of Formatting Objects (FO). The FO description can be thought of as a formatter independent layout description. Then an FO processor is used to produce actual printed output, on paper or as an electronic document.

XSL stylesheets for Docbook have been available for several years, written by Norman Walsh. They implement a large part of the Docbook elements—not all elements, that seems impossible. And they are extensively parametrized, so that users can customize many aspects without modifying the XSL code.

The objective of XSL + FO is: one stylesheet, many processors. Several FO processors are available, among which two free tools: FOP and TEX. FOP is a dedicated FO processor, that produces output in PDF. It is available from the Apache website[4]. TEX can be used as an FO processor using David Carlisle's `xmltex` and Sebastian Rahtz's `passivetex` package, which runs under LaTEX.
ConTEXt is a prospective FO processor. It already has an XML parser. Mappings should be defined for Formatting Objects, in the same way as I have done for Docbook In ConTEXt.

Future plans

Currently, Docbook In ConTEXt is not completely integrated with the ConTEXt distribution. I have strictly used the ConTEXt API wherever I could, and avoided to develop my own variants. But I have preferred to develop this module in separation from the development of ConTEXt itself. The time has now come to work on a better integration. I hope this can be achieved over the next year.

If good, customizable XSL stylesheets for Docbook exist, and if ConTEXt could be an FO processor for the resulting output, then why would it be a good idea to spend so much effort on writing a special Docbook stylesheet for ConTEXt?

In the ConTEXt community the idea of a special Docbook stylesheet for ConTEXt has been greeted with enthousiasm. Apparently, here the theory of one stylesheet for many processors succumbs to the practice that users prefer to work with their tools of choice. For a popular set of tools like Docbook and ConTEXt users afford the effort of another style sheet. Such a style sheet is more manageable for them and running the required tools is easier.

On the other hand, until now, *I* have spent most of the required effort. And *my* answer tends to be: Maybe it is *not* the best way to support Docbook and XML in ConTEXt. Maybe it would be more useful to work on FO mappings in ConTEXt.

Over the past year I have set up this stylesheet. I have investigated the main structure of Docbook and come up with a way to map that to a ConTEXt document. I have implemented a framework for the mapping. I have enjoyed doing all that, and my insight and skills in TEX macro programming have increased immensely. But the time has come that others take this over, add mappings for more elements, add customizations, add new ideas. I plan to move forward to more generic work to support formatting of XML documents using TEX as the typesetting tool.

Availability

Currently, Docbook In ConTEXt is available separately from the ConTEXt distribution, from my web site[5]. Michael Wiedmann's web page[6] with Docbook tools has a link to the Docbook In Context files.

Programming Docbook In ConTEXt

ConTEXt and XML

ConTEXt can take XML documents as input. For that purpose it contains a non-validating XML parser, which recognizes XML tags as markup instructions. And it has an API

4. `http://xml.apache.org/fop`
5. `http://www.hobby.nl/~scaprea/context`
6. `http://www.miwie.org/db-context/index.html`

(Application Programmer's Interface) which allows one to define actions for those tags. This is called mapping XML tags to ConTEXt. A typical mapping instruction is

```
\defineXMLenvironment[element]{start action}{stop action}.
```

During the start and stop actions one has access to the attribute values of the element. For example, this is how one reads the `align` attribute of an `entry` element (in a table) and issues the corresponding setup command for ConTEXt's `TABLE` environment:

```
\doifXMLvar{entry}{align}%
   {\expanded{\setupTABLE[align=\XMLvar{entry}{align}{}]}}
```

ConTEXt's programming interface for XML mapping is robust. Rarely if ever does one get tangled in expansion problems. But, as is seen in the above example, timing the expansion remains an issue: The command to retrieve the attribute value, `\XMLvar{entry}{align}{}`, must be expanded before the setup command can be read by TEX. That is what `\expanded` does.

It is easy, is it not?

In principle, writing a mapping for an XML document in ConTEXt is simple. You state which ConTEXt commands you want to use for the start and stop of each element, and ConTEXt takes care of the rest. Practice is more complicated, certainly if you want to write a useful, extensible and customizable mapping for a complicated DTD. In the following sections I discuss a number of noteworthy features of the Docbook In ConTEXt mapping.

Encoding and language

An XML document declares its encoding in the xml declaration at the start of the document. ConTEXt supports several encodings, among which the XML default encoding `utf-8`. Correctly reading an encoding is one thing. Making all characters available that can be addressed by an encoding is quite another thing. Unicode and its utf-8 encoding have brought all characters in the Unicode range, currently more than 50,000, within scope in a single document. At the moment many of these are mapped to 'unknown character'. Work is ongoing to bring more characters within reach of ConTEXt in a single document.

A Docbook document may declare its language in the `xml:lang` attribute of the document element. The Docbook in ConTEXt module contains at the moment translated strings for four languages: English, German, Dutch and Italian. These are used for automatically generated strings, such as the titles of the table of contents, the abstract, and the index.

Features for each element

Context stack

Because an XML document has a tree structure, each element in the document has a list of ancestors. I call that the context, or the context stack, which contains the ancestors from the document root to the current element.

An element may push itself onto the context stack when it starts, and pop itself when it finishes. In principle all elements should do so. In practice a number of elements omit this because they or their children do not use the context stack in their formatting.

During formatting, the context stack can be inspected with the following commands:

- `\XMLDBcurrentelement`: The current element's name.
- `\XMLancestor#1`: The name of the ancestor at level `#1` The current element is at level 0.
- `\XMLparent`: The name of the current element's parent.

- \the\XMLdepth: The depth of the context stack.
- \doifXMLdepth#1: Execute the following instruction if the context stack has a certain depth.
- \XMLDBprintcontext: Print the context stack in the log file (mainly for debugging purposes).

ConTEXt also defines the context stack. I have redefined it because ConTEXt's implementation did not satisfy my plans. Later I have simplified my usage of the context stack. ConTEXt's implementation may now be perfectly satisfactory, but I have not checked this.

ConTEXt defines \currentXMLelement, which also holds the name of the current element. But it is only guaranteed to be valid while ConTEXt reads the XML tag. Indeed, the mapping of some start tags in Docbook in ConTEXt emit an \egroup command, which invalidates the value of \currentXMLelement.

Ignorable white space

XML has the interesting feature of ignorable white space. It can be used to give the raw XML document a nice formatting and make it fairly readable. (It did not exist in SGML. As a consequence, SGML documents may be practically unreadable in an ASCII editor.) For applications that read the DTD, this feature is rather clear: white space in elements whose content may only consist of elements, is ignorable. For example, when the content model of a section only contains paragraphs, all white space that surrounds the paragraphs is ignorable. Applications like ConTEXt that do not read the DTD, must resort to other means to find out whether white space is ignorable or not.

I have introduced a feature that is similar to the mechanism used in XSLT. One can declare that an element preserves white space with the command \defineXMLDBpreservespace#1, and that it ignores white space with the command \defineXMLDBstripspace#1. For these declarations to work, the elements should be on the context stack, and they and their children should use the command \XMLDBdospaces as the last command in their start and end tags. \XMLDBdospaces has the effect of ignoring spaces following the XML tag if the current element has been declared to ignore spaces.

In practice this is only used by elements that would suffer if white space is not ignored. Note that TEX itself already ignores a lot of white space, viz. all white space that it reads in vertical mode. In the example of white space surrounding paragraphs in a section, TEX would do the right thing by itself.

The correct functioning of \XMLDBdospaces is rather subtle. The following is a generic element mapping:

```
\defineXMLenvironment[xxx]
   {\XMLDBpushelement{\currentXMLelement} \XMLDBdospaces}
   {\XMLDBpopelement \XMLDBdospaces}
```

The command \XMLDBdospaces in the start tag is executed while xxx is the current element. So it ignores white space if xxx has been declared to contain ignorable white space. But the same command in the end tag is executed *after* xxx has popped itself from the context stack. So its parent is the current element, and the command ignores white space if that *parent* has been declared to ignore white space. That is indeed exactly what we want, because the spaces following the end tag </xxx> are in the parent's content.

There is a class of ignorable white space that TEX refuses to ignore: blank lines are converted to \par commands by TEX's input scanner, before we can tell TEX whether white space is ignorable or not. Even this does not always matter to TEX because TEX discards empty paragraphs or paragraphs that consist of white space only. In the above example we could insert blank lines between the paragraphs without ill effect. But a blank line between the start tag of a footnote and its first paragraph has a notably bad

effect: it introduces a \par command between the footnote number and the start of the text, so that the footnote number is in a paragraph by itself.

Such harmful blank lines can only be removed by preprocessing of the XML document. I wrote a tool to do that. It is a SAX document handler written in Java, which removes all ignorable white space. I call it 'Normalizer', and it is available on my web site[7].

The output of this tool is not only good for the ConTEXt mapping. Looking over it is informative for authors of XML documents. Every amount of white space that is left by the tool, is regarded as meaningful white space by XML parsers. Is that really what the author wants?

Every element

In principle every element should contain the following commands:

```
\defineXMLenvironment[xxx]
   {\XMLDBpushelement\currentXMLelement
    \XMLDBseparator \XMLDBdospaces}
   {\XMLDBpopelement \XMLDBdospaces}
```

That is, it pushes itself onto the context stack. It checks whether it should typeset a separator. And it checks whether it should ignore following white space. In its end tag, it pops itself from the context stack, and it checks whether its parent should ignore following white space.

The separator is used by such elements as author, which may generate a comma or the word 'and' between consecutive elements. By default it is set to \relax. A parent element should give it a suitable definition to be used by its children, and reset it to the default when it finishes.

Which element is next?

ConTEXt's XML parsing is event based. This means that the parser generates events, such as the start or stop of an element, and calls the associated actions. During the actions one only sees the current event. One cannot look back at past events, except for the data that one saved. One can certainly not look forward to check which elements follow. In contrast, XSLT is tree based. That means that one can scan all elements, preceding and following, in the formatting commands of an element. Event-based parsing may present serious problems to the programmer.

Is there a title?

An abstract may but need not have a title. When there is no title, I want to print the default title 'Abstract'. Because of the event-based nature of the parse, one cannot at the start of the abstract look forward to see if a title will follow. One can only try to find a future event at which one may safely conclude that there is no title if one has not yet seen a title.

In an abstract the optional title may be followed by three types of element, `para`, `simpara` and `formalpara`. When any of these elements is started, one may safely conclude that either the title has been seen or there is no title.

One solution is to save the title, and to redefine the mappings of each of these three elements, such that they output the title or the default title if there was no title. And then restore their default definitions for the following elements.

Another way to tackle this problem is to save the whole abstract and process it twice. In the first pass we check whether there is a title. During this pass, all output should be suppressed. In the second pass we first output the title or the default title if no title was

7. http://www.hobby.nl/~scaprea/context

found in the first pass, and then we output the content. Again this requires a redefinition of the three possible elements that may follow the title, so that they suppress their output at the first pass.

The third option is provided by TEX itself, not by the XML mapping. We redirect the typesetting of the abstract into a `vbox`. At the same time we save the title in the variable `\XMLDBtitletext`, which removes it from the typeset content in the `vbox`. Then we output the saved title or the default title if there is no saved title, and next we output the `vbox`. This is the best option, and I use it.

In a section this solution would be more problematic: we run the risk of saving a large `vbox`. Working with options one or two is also not fun, because there are more elements to be redefined. I think the only viable alternative would be to work with `\everypar`, because `\everypar` is TEX's low-level signal that there is new text. Fortunately, in a section a title is required, so I did not (yet) have to work out this problem.

This is an example of the problems that arise because in an event-based parse it is hard to determine if an optional element is not present. The following section presents an example of the problems that arise because in an event-based parse it is equally hard to determine when a certain group of elements is finished.

Sectioning

Like many systems, ConTEXt partitions its document in frontmatter, bodymatter, appendices and backmatter (called section blocks). The section block governs such properties as the numbering of the chapters and sections. I use the end of the frontmatter to print the table of contents.

Docbook does not have the equivalent of section blocks. There is not a single element that contains the frontmatter, the bodymatter or the backmatter. Therefore I analysed the top-level structure of a docbook document, and divided the elements that may occur as top-level elements into frontmatter elements, bodymatter elements and backmatter elements. When the first top-level bodymatter element is seen, the frontmatter is complete and the bodymatter starts. Similarly for the backmatter.

For a book in Docbook the situation is rather clear: The bodymatter starts with the first `part`, `chapter`, `article` or `reference`.

For an article the situation is much more fuzzy. While I counted only 6 top-level frontmatter elements, I identified 60 top-level bodymatter elements. The situation is complicated by the fact that some of these 60 top-level bodymatter elements may occur in the frontmatter at a lower level. For example, `abstract`, `authorblurb` and `address` may occur within the element `articleinfo`, which itself is a main constituent of the frontmatter. They may also occur as top-level elements, in which case I consider them as part of the bodymatter.

The transitions between the other section blocks are fortunately more clearly marked. The complete analysis is contained in the documentation in the module itself.

The situation is programmed using the commands `\XMLDBmayensurebodymatter` and `\XMLDBmayensurebackmatter`. All top-level bodymatter and backmatter elements execute the appropriate command. These commands check if the element is a direct child of the document element, i.e. if the depth of the context stack equals 2, and if the corresponding section block has not yet been started. The current section block is kept in the variable `\XMLDBsectionblock`.

Specific elements

Tables

Docbook uses the CALS table model. ConTEXt uses two different table models. One is the tabulate environment, which is based upon TEX's `\halign`. It is quite sensitive to expansion timing errors. The other is the TABLE environment, also called natural tables. It

is a very powerful and flexible environment, with much customization possibilities using \setupTABLE commands. A special feature of this table model is that rows, columns and cells can be configured both before and after their content has been given, at any time before the end-of-table (\eTABLE) command.

Because ConTEXt's natural tables have much similarities to CALS tables, the mapping is in principle very easy: a row corresponds to TR, an entry to TD, colspec elements can be mapped to \setupTABLE commands.

There are three main complications.

- The top, bottom, left and right frames of a CALS table are determined by the frame attribute of the table; the rowsep and colsep attributes of the corresponding rows and cells should be ignored.
- CALS tables can have multiple tgroup elements, each with their own number of columns, and their own alignment and frame settings (colspec elements).
- Each tgroup may have its own thead and tfoot elements, which may contain their own colspec elements.

These requirements have led to the following model: The table element generates a ConTEXt table, i.e. the table float, using the \placetable command. Each tgroup element generates its own TABLE environment, i.e. the actual table.

The table is not openend by the start tag of the table, because at that moment the title is not yet known. Instead, it is opened by the start tag of the first tgroup (command \XMLDBopentable, which contains the ConTEXt command \placetable). The start tag of each following tgroup typesets the previous tgroup (command \XMLDBendTABLE). Before typesetting, the left and right frames are set up. The start tag of the second tgroup also sets up the top frame. The end tag of the table does the same as the start tag of the next tgroup would do. In addition it sets up the bottom frame of the table, and closes the vbox of the \placetable command.

The rest is careful attribute processing, and issuing the required \setupTABLE commands at the right time. Attribute processing generates a lot of overhead, because both the attribute names and their possible values have to be translated from CALS to \setupTABLE. That makes the code somewhat less readable, but the logic is quite straightforward.

Issuing the required \setupTABLE commands is a precise work.

- The start tag of the first tgroup applies the frame, colsep and rowsep attributes of the table (\XMLDBopentable), so that they apply to all TABLEs in this CALS table. The start tag of each tgroup applies its own align, colsep and rowsep attributes, within its own TABLE environment.
- colspec elements of a tgroup apply their attributes to the whole column of this TABLE. The colspec elements in the thead and tfoot elements, on the other hand, must save their attributes (\XMLDBsavecolspec); they will be applied per entry in the thead and tfoot.
- row elements apply their attributes immediately to the whole row.
- entry elements first check whether they are in a thead or tfoot; if so, they apply the saved colspec attributes. Then they apply their own attributes. This order is important. ConTEXt gives precedence to properties set up per cell over properties set up for the whole table or per row or column. But in this case we apply what was originally a column specification per cell, so we must take care of the precedence ourselves.

Revision history

The revision history contains a number of revisions. Each revision specifies one or more fields out of five possible fields. I wanted to represent this in a table which should only

contain those columns for which at least one revision specifies data. Programming this was my first challenge in this project.

Hans Hagen suggested the solution. The revision history is saved, and then processed twice.

For the first pass of the saved revision history, we define the revision fields such that they register themselves when they occur, but suppress all output. We also count the number of revisions, so that we will know which row must contain the bottom rule of the table. Now we know which fields occur and we can setup the table and output its header row.

For the second pass we define the revision element such that it outputs the row with the fields. So that the fields are output in the same order as in the header row, regardless of their order in the XML document, we first save the fields in a revision, and at the end tag of the revision we output the whole row in the desired order.

I worked this out both in ConTEXt's tabulate environment and with its natural tables. I decided to keep the solution with the natural tables, because natural tables are more flexible and less prone to expansion errors.

This procedure demonstrates a powerful feature of ConTEXt's XML processing: It is possible to save a node of the XML document with its subtree; in other words, the content of an element, complete with embedded elements, is saved in a variable without parsing. Later one can process the saved subtree as often as one likes. In between one is free to redefine the behaviour of the embedded elements. In TEX's macro language this is quite normal behaviour :

```
\def\savevar#1{\def\var{#1}}
... % redefine \processvar
\processvar{\var}
... % redefine \processvar
\processvar{\var}
```

In other programming languages it is not nearly as easy. Saving a node with its subtree in a SAX content handler so that it can be processed later is not a trivial task.

It is a disadvantage of the above procedure that the code is not easily read, certainly not if one is not used to the procedure. Recently, I have discussed an alternative procedure using Giuseppe Bilotta's `xdesc` module. It would achieve the same result but make the programming more transparent. Another advantage would be that it is more easily customizable by the user.

Program listing and CDATA

I have spent an enormous amount of time on program listings. At first it seemed easy: ConTEXt has a verbatim environment which suits our purpose.

Then it was pointed out to me that some program listings contain CDATA sections, which were not treated well by my solution. I realized that a program listing is not really a verbatim environment because it does not disable XML tags. I dived deep into ConTEXt's verbatim environment and came up with a variant that supported two types of verbatim: one real verbatim for `CDATA` sections and one that did only line oriented layout for program listings. Moreover, it was nestable, so that it could deal with `CDATA` sections within program listings.

But it remained problematic to get it quite right. When the end of the `CDATA` section or of the program listing element was followed by text on the same line, this text was lost. And my white space tool did exactly that: put the following text right behind the end of the program listing element.

When I revisited the problem a few months later it dawned on me that the whole verbatim approach was wrong. Neither `CDATA` sections nor program listing environments have anything to do with TEX's notion of verbatim. `CDATA` sections just disable XML

markup. They may occur anywhere in an XML document, and have no semantic meaning. Indeed, an XML parser does not even report whether `CDATA` sections are used in an XML document; it simply resolves them.

For the program listing I found a simple solution. It avoids scanning a whole line at a time, therefore it avoids scanning the text following the end of the program listing with the wrong catcodes in place. It uses `\obeylines` and `\obeyspaces` and it places struts at the start of a line to prevent the leading spaces to be discarded by TEX's paragraph mechanism. That is all, and it does the job well.

Hyperlinks, URLs and external documents

Docbook documents mark hyperlinks with the `ulink` element; the url is contained in its `url` attribute. If we were writing HTML documents it would be easy: `<ulink url="URL">text</ulink>` would be translated to `<a href="URL">text</href>` and the browser would do the rest.

But not such an easy solution in a PDF document. Links to PDF documents should be treated differently from links to other documents, and relative links to non-PDF documents are not allowed. Therefore, we have to analyse the URL and complete it if necessary.

In ConTEXt strings can be split into parts with commands like

```
\beforesplitstring string\at substring\to\var
```

which splits `string` at `substring` and stores the first part in `\var`. I use this and similar commands to check whether the URL has an authority (this is the term used by RFC2396, which specifies URIs; usually it is called the protocol, e.g. `http`) and whether it is an absolute or a relative URL. If a local file is specified, we also check whether it has the extension `pdf`. Links to local PDF documents are created using the ConTEXt command `\useexternaldocument`, links to other documents use the ConTEXt command `\useURL`.

A special problem is posed by URLs like `slashdot.org`. Is it a web server, or is it a file in the current directory? Cf. the URL `myfile.html`, which has exactly the same pattern. After the terminology of RFC2396 I call this abbreviated URLs. By default they are not recognized. Thus `myfile.html` is correctly linked to as a local document, while `slashdot.org` is incorrectly linked similarly. The user can switch recognition of abbreviated URLs on by setting `\XMLDBcheckabbrURItrue`, and can switch if off again by setting `\XMLDBcheckabbrURIfalse`.

Unfortunately, I do not know how to get the working directory in a ConTEXt run, so that relative URLs are currently not properly completed.

Customization

For a long time I did not pay much attention to customization. Recently, I received requests to make a mapping for the `blockquote` and `epigraph` elements. Together with that request a discussion arose on the ConTEXt mailing list about customization. As a consequence these two elements and their child element `attribution` have proper setup options, viz. the commands `\setupblockquote`, `\setupepigraph` and `\setupattribution`. I am sure that more such setup commands will follow.

The same discussion on the ConTEXt mailing list touched upon attributes whose range of values is not constrained. An example is the `role` attribute of the `para` element. It is not possible to define actions for such attributes in the stylesheet, because the possible values are not known. The idea arose to put a hook in the stylesheet for the user's own formatting command. Something like `\attributeaction[para][role]`. The user could define such an action with something like `\defineXMLattributeaction[para][role]`.

The stylesheet should invoke the attribute action within a group in order to allow the user to change fonts etc. for this element. Therefore ConTEXt cannot invoke the attribute action automatically, because it cannot know where it should do so. For example, some

mappings for the opening tag invoke `\egroup`; if the attribute action had been invoked automatically, its scope would be ended immediately. This idea has not yet been implemented.

I am not sure how far customization can go. Enabling extreme customizability would come down to defining a new language for describing the formatting of a Docbook document. This would go too far. On the other hand, customizability is a strong feature of ConTEXt. It is not difficult to add customizability options to the stylesheet; ConTEXt has some good commands for that.

Acknowledgement

The first versions of the mappings for several elements, a.o. the `ulink` element, were contributed by Michael Wiedmann. He also contributed the string literal files for English and German. Giuseppe Bilotta contributed the string literals file for Italian.

And of course, nothing of this would have been possible without Hans Hagen's ConTEXt.